# OptiProfiler: A Benchmarking Platform for Optimization Solvers

Cunxin Huang[*] Tom M. Ragonneau[†] Zaikun Zhang[‡]

February 4, 2026 3:07am +08:00

**Abstract**

We present OptiProfiler, a benchmarking platform for optimization solvers, with a current focus on derivative-free optimization (DFO). OptiProfiler provides automated, flexible tools for evaluating solver performance, supporting performance profiles, data profiles, and log-ratio profiles. The platform enables users to benchmark their solvers on a variety of testing environments by providing several built-in features and problem libraries, while allowing users to customize their own.

**Keywords**: Automated benchmarking, derivative-free optimization, performance profiles, data profiles, log-ratio profiles

## 1 Introduction

Benchmarking algorithms and solvers is essential in optimization and has attracted increasing interest in the derivative-free optimization (DFO) community [1, 4, 13]. Despite its importance, existing benchmarking tools for DFO are limited in automation, usability, and flexibility. For example, performance profiles [5] and data profiles [15] are widely used, but they are typically implemented using the basic code provided in [5] (see `https://www.mcs.anl.gov/~more/dfo`), which is aimed at experts. Therefore, there is a clear need for a more comprehensive and user-friendly platform that can facilitate reproducible and insightful benchmarking.

Several benchmarking platforms and tools have been developed. For example, the COCO platform [11] is widely used for evolutionary algorithms. However, COCO lacks

[*]Department of Applied Mathematics, The Hong Kong Polytechnic University, Hong Kong, China (`cun-xin.huang@connect.polyu.hk`).

[†]Axians HPC, company of VINCI Energies, Toulouse, France (`tom.ragonneau@gmail.com`).

[‡]School of Mathematics, Sun Yat-sen University, China (`zhangzaikun@mail.sysu.edu.cn`).

support for performance profiles and is not well suited for benchmarking on classical problem libraries (or collections) such as CUTEst [7]. A recent technical report [2] by the research group from Polytechnique Montréal and GERAD describes their toolbox "RunnerPost" for post-processing and profiling existing optimization results; however, it is not fully automated and can be difficult for non-experts to use.

Our goal is to provide a reliable, extensible, and easy-to-use benchmarking platform, beginning with a focus on DFO solvers. OptiProfiler addresses this need by supporting automated benchmarking and customizable testing environments, including a variety of features (such as adding noise or perturbing initial points) and problem libraries to evaluate solver robustness under different conditions. By offering these capabilities, OptiProfiler aims to help researchers and practitioners focus on algorithm development and analysis, while ensuring that benchmarking is easy, rigorous, and reproducible. The official website of OptiProfiler is available at

`https://optprof.com`.

The source code and documentation are available on GitHub at

`https://github.com/optiprofiler/optiprofiler`.

This paper is organized as follows. In Section 2, we provide an overview of OptiProfiler through a simple example. In Section 3, we describe how we define the testing environments in OptiProfiler, which consist of problems and features. In Section 4, we describe the benchmarking tools implemented in OptiProfiler, including performance profiles, data profiles, and log-ratio profiles; we also score solvers based on these profiles. The equivalence of performance profiles and data profiles is discussed in Section 5. Section 6 presents two demonstration experiments: one benchmarks representative solvers, and the other tunes the hyperparameters of a solver. Finally, we conclude the paper and outline future extensions in Section 7.

## 2 Overview of OptiProfiler through a simple example

OptiProfiler is designed to simplify and automate the benchmarking process for optimization solvers by providing flexible tools and insightful visualizations. To illustrate its core functionalities, this section presents a simple example.

Suppose we want to benchmark two standard MATLAB solvers, `fminsearch` (the Nelder–Mead simplex method [12, 16]) and `fminunc` (a finite-difference BFGS method when the gradient is not provided), on the default testing environment. This can be achieved with the following MATLAB command.

```matlab
% Benchmark simplex method and BFGS with default testing
  environment.
```

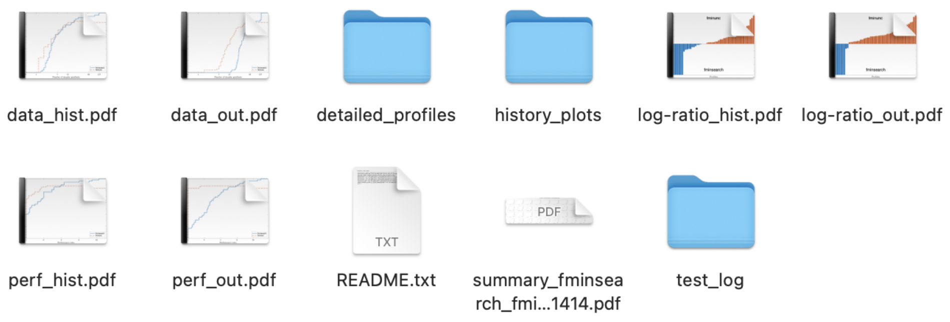


Figure 1: Screenshot of the folder containing the results of the benchmark.

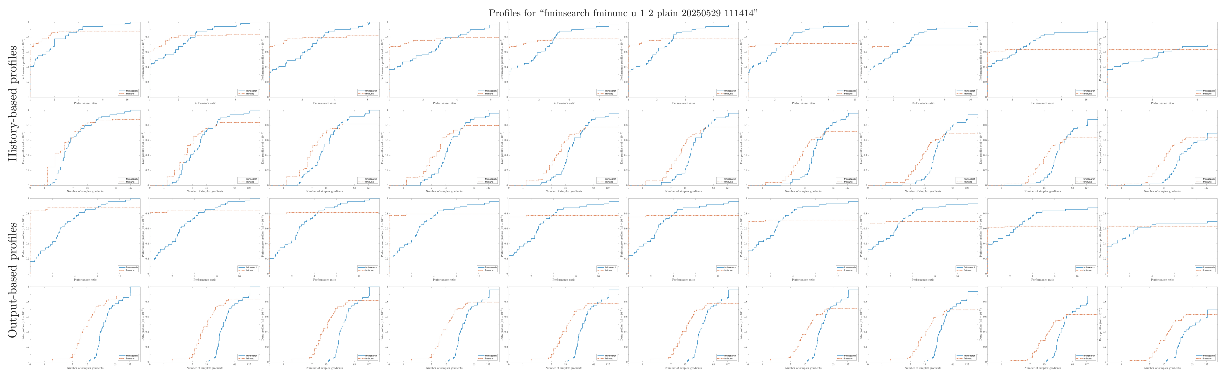


Figure 2: Screenshot of the summary PDF of all the performance profiles and data profiles.

```
benchmark({@fminsearch, @fminunc})
```

This command benchmarks the solvers on unconstrained problems from the default problem library with dimension at most 2, using the default feature `plain` (see Section 3 for more details). OptiProfiler automatically generates a folder named `out` in the current directory. It contains a subfolder named with an experiment timestamp (see Figure 1 for a screenshot) that includes all detailed results. A PDF file named `summary.pdf` (see Figure 2 for a screenshot) is also generated and saved in `out`; it summarizes all performance profiles and data profiles on a single page. Note that `fminsearch` and `fminunc` can be replaced by any callable functions that accept the same basic signature (e.g., `x = fminsearch(fun, x0)`, where `fun` is the objective function, `x0` is the initial guess, and `x` is the returned solution).

This example demonstrates how OptiProfiler enables comprehensive benchmarking with minimal effort. Next, we introduce the testing environments used in OptiProfiler, which consist of optimization problems and features.

## 3 Major components: problems and features

Testing environments are a core component of OptiProfiler, enabling users to evaluate solvers on a wide range of optimization problems under various conditions. A testing environment consists of two parts: the optimization problems to be solved and the features applied to these problems. By combining problems and features, OptiProfiler enables a systematic assessment of solver performance across diverse scenarios.

In Subsection 3.1, we describe the optimization problem we consider, how it is represented, and how features are defined. In Subsection 3.2, we describe the problem library provided by OptiProfiler.

### 3.1 Problem, feature, and featured problem

We consider the following optimization problem:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & x_\mathrm{l} \le x \le x_\mathrm{u}, \\ & A_\mathrm{ub} x \le b_\mathrm{ub}, \\ & A_\mathrm{eq} x = b_\mathrm{eq}, \\ & c_\mathrm{ub}(x) \le 0, \\ & c_\mathrm{eq}(x) = 0, \end{aligned} \tag{3.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $x_\mathrm{l}, x_\mathrm{u} \in \mathbb{R}^n$ are the lower and upper bounds, $A_\mathrm{ub}, b_\mathrm{ub}, A_\mathrm{eq}, b_\mathrm{eq}$ define linear constraints, and $c_\mathrm{ub}, c_\mathrm{eq}$ denote nonlinear constraints.

To represent the optimization problem (3.1), OptiProfiler defines a class named `Problem`. This class serves as the foundation for defining and managing optimization problems in a structured and flexible way. It encapsulates all the necessary information about a problem, including its properties (e.g., initial guess, constraints) and methods (e.g., objective function evaluation). For readers unfamiliar with object-oriented programming in MATLAB, a property is a data value that belongs to the class (similar to a variable that stores information), while a method is a function that can be performed on or by the class. Below, we provide an overview of the key components of the `Problem` class.

The `Problem` class contains three types of properties.

- **Essential properties.** These must be provided when defining a problem. Specifically,
  - `x0`: the initial guess for the variables $x$;
- **Optional properties.** These provide additional details about the problem, such as
  - `name`: the name of the problem;

  - `xl`, `xu`: the lower and upper bounds on the variables;
  - `aub`, `bub`: the matrix and vector defining linear inequality constraints.

- **Dependent properties.** These are automatically computed based on the provided information. Examples include
  - `ptype`: the type of the problem, which can be `'u'` (unconstrained), `'b'` (bound-constrained), `'l'` (linearly constrained), or `'n'` (nonlinearly constrained);
  - `n`: the dimension of the problem.

The `Problem` class also provides several methods for evaluating the problem, including the following.

- `fun`, `grad`, and `hess`: the objective function $f$ in (3.1), and its corresponding gradient and Hessian if available;
- `cub`, `ceq`, `jcub`, `jceq`, `hcub`, and `hceq`: the nonlinear constraints $c_{\text{ub}}$ and $c_{\text{eq}}$ in (3.1), and their corresponding Jacobian and Hessian if available;
- `maxcv`: the maximum constraint violation at a given point $x$, which is defined as the maximum of $\|(x_{\text{l}}-x)^{+}\|_{\infty}$, $\|(x-x_{\text{u}})^{+}\|_{\infty}$, $\|(A_{\text{ub}}x-b_{\text{ub}})^{+}\|_{\infty}$, $\|A_{\text{eq}}x-b_{\text{eq}}\|_{\infty}$, $\|(c_{\text{ub}}(x))^{+}\|_{\infty}$, and $\|c_{\text{eq}}(x)\|_{\infty}$.

Note that only `fun` is essential among these methods; the others are optional.

Despite testing solvers on problems in the form of (3.1), people may be interested in benchmarking solvers on problems with different "features", e.g., adding noise to the objective function. Thus, we define a class `Feature` and a subclass of `Problem` named `FeaturedProblem`. A `Feature` is a mapping that modifies a `Problem` to create a `FeaturedProblem`, which inherits all properties of the original problem but includes additional modifications (e.g., adding noise or perturbing the initial guess). The `FeaturedProblem` class also records the computation history of solvers on the modified problem, enabling detailed analysis of solver behavior.

OptiProfiler provides a set of built-in features.

- `plain`: do not modify the original problem;
- `perturbed_x0`: randomly perturb the initial guess $x_0$;
- `noisy`: add noise to the objective function and nonlinear constraints;
- `truncated`: truncate the objective function and nonlinear constraints to a given precision;
- `permuted`: randomly permute the order of the variables;

- `linearly_transformed`: apply a linear transformation to the variables using the product of a positive diagonal matrix and a random orthogonal matrix;
- `random_nan`: randomly set some values of the objective function and nonlinear constraints to NaN;
- `unrelaxable_constraints`: set the objective function to infinity outside the feasible region;
- `nonquantifiable_constraints`: replace values of nonlinear constraints with either 0 (if satisfied) or 1 (if violated);
- `quantized`: quantize the objective function and nonlinear constraints.

To use these features, users can simply pass the corresponding feature name to the `benchmark` function as the second argument or as a field of a structure. For example, if we continue to use the previous example and want to try the `noisy` feature, we can use the following command.

```
% Method 1.
benchmark({@fminsearch, @fminunc}, 'noisy')
% Method 2.
options.feature_name = 'noisy';
benchmark({@fminsearch, @fminunc}, options)
```

The built-in features also provide several customization options. For example, the `noisy` feature allows users to specify the noise level, noise type (absolute, relative, or mixed), and the distribution of the noise. If we want to try the `noisy` feature with an absolute noise following a uniform distribution and a noise level of 0.1, we can use the following command.

```
% Test customized 'noisy' feature.
options.feature_name = 'noisy';
options.noise_type = 'absolute';
options.distribution = 'uniform';
options.noise_level = 0.1;
benchmark({@fminsearch, @fminunc}, options)
```

## 3.2 Problem library

OptiProfiler provides a default problem library based on the CUTEst library [7]. The CUTEst library has been developed over many years and integrates a wide range of high-quality unconstrained and constrained test problems, including both research-oriented and real-world problems. It has become a widely used standard for benchmarking optimization solvers, and as mentioned in [15], "most researchers have relied on a selection of problems from the CUTEr [6]" (the predecessor of CUTEst). The version of CUTEst we use is the latest S2MPJ library [10], which is implemented entirely in MATLAB, Python, or Julia, and can be used directly without any additional installation or interfacing with MEX files or Fortran programs. For Linux systems, we also provide support for MatCUTEst [21], a MATLAB interface to CUTEst that requires MEX files. Using MatCUTEst can significantly accelerate computations due to the use of precompiled binaries.

To select a specific problem library and filter problems, users can specify options in the options structure and pass it to the `benchmark` function. For example, to use problems from both S2MPJ and MatCUTEst that are either unconstrained or linearly constrained, with dimension less than 10 and no more than 20 linear constraints, use the following commands.

```
% Select problems from libraries.
options.plibs = {'s2mpj', 'matcutest'};
options.ptype = 'ul';
options.maxdim = 10;
options.maxlcon = 20;
```

Further options for selecting and filtering problems are available; see the documentation for a comprehensive list.

While CUTEst provides a comprehensive and well-established library of test problems, it also has some limitations. For example, CUTEst is primarily focused on smooth problems with continuous variables and may not cover certain classes of problems such as nonsmooth or mixed-integer optimization. This will not be an issue since OptiProfiler is designed to be flexible and extensible, making it straightforward to use other problem libraries or custom libraries. By adhering to the `Problem` class interface, users can easily integrate new problem libraries, including those tailored to specific application domains or problem structures, thus extending the benchmarking capabilities beyond the default libraries.

## 4 Benchmarking tools

## 4.1 Definitions of profiles

We review the definitions of performance profiles [5, 15], data profiles [15], and log-ratio profiles [19], and introduce the history-based and output-based variants implemented in OptiProfiler.

Let $\mathcal{P}$ denote a set of test problems and $\mathcal{S}$ a set of solvers. For each problem $p \in \mathcal{P}$ and solver $s \in \mathcal{S}$, let $t_{p,s}$ denote the absolute cost incurred by solver $s$ when solving problem $p$ up to a convergence test (see (4.3) in Subsection 4.2). The absolute cost is referred to as a performance measure in [5, 15] and can, for example, be computing time or the number of function evaluations. In this paper, we focus on the case where the number of function evaluations is used as the absolute cost.

### 4.1.1 Performance profiles

Define the relative cost of solver $s$ on problem $p$ as

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in \mathcal{S}\}},$$

with the convention that $\infty/\infty = \infty$. The performance profile [5, 15] of solver $s$ is defined as

$$\rho_s(\alpha) = \frac{1}{|\mathcal{P}|} \left|\{p \in \mathcal{P} : r_{p,s} \le \alpha\}\right| \quad \text{for } \alpha \ge 1, \tag{4.1}$$

where $|\cdot|$ denotes the cardinality of a set. The performance profile $\rho_s(\alpha)$ is the fraction of problems in $\mathcal{P}$ that are solved by solver $s$ with a relative cost at most $\alpha$.

Performance profiles are widely used in optimization, and as mentioned in [3], "have become a gold-standard in modern optimization benchmarking, and should be included in optimization benchmarking analysis whenever possible with an appropriate interpretation." However, readers should be aware of the limitations pointed out in [8] that "we cannot necessarily assess the performance of one solver relative to another that is not the best." When comparing multiple algorithms, it is often beneficial to examine the pairwise performance profiles between each pair of solvers in addition to the overall profiles.

### 4.1.2 Data profiles

The data profile [15] of solver $s$ is defined as

$$\delta_s(\alpha) \;=\; \frac{1}{|\mathcal{P}|}\left|\left\{p \in \mathcal{P} : \frac{t_{p,s}}{n_p + 1} \leq \alpha\right\}\right| \quad \text{for } \alpha \geq 0, \tag{4.2}$$

where $n_p$ is the dimension of the problem $p$. Note that the data profile may be understood in the same way as the performance profile in (4.1) except that $r_{p,s}$ is changed to

$$r_{p,s} \;=\; \frac{t_{p,s}}{n_p + 1},$$

which can also be interpreted as the number of simplex gradients [15] if $t_{p,s}$ is the number of function evaluations.

Data profiles are useful for comparing solvers on problems of different dimensions, as they normalize the absolute cost by the number of variables. This normalization allows for a fair comparison across problems with varying sizes, making data profiles particularly valuable in benchmarking scenarios where problem dimensions differ significantly.

### 4.1.3 Log-ratio profiles

When $\mathcal{S}$ only contains two solvers $s_1$ and $s_2$, we denote the log-ratio of the absolute costs of the two solvers on problem $p$ as

$$l_p \;=\; \log_2\left(\frac{t_{p,s_1}}{t_{p,s_2}}\right) \quad \text{for } p \in \mathcal{P}\backslash\mathcal{E},$$

where $\mathcal{E}$ is the set of problems that both solvers fail to solve, i.e.,

$$\mathcal{E} \;=\; \{p \in \mathcal{P} : t_{p,s_1} = t_{p,s_2} = \infty\}.$$

The log-ratio profile is defined as the sequence

$$\{\lambda_1, \ldots, \lambda_{|\mathcal{P}|-|\mathcal{E}|}\} \;=\; \operatorname{sort}\left(\{l_p : p \in \mathcal{P}\backslash\mathcal{E}\}\right),$$

where $\operatorname{sort}(\cdot)$ arranges the numbers in ascending order.

Note that the case $t_{p,s_1} = t_{p,s_2} = \infty$ is not discussed in [14, 19], since $t_{p,s}$ is always finite by setting a maximum budget of function or gradient evaluations. Log-ratio profiles are particularly useful for comparing two solvers, as they provide a clear visualization of the relative performance of the two solvers. OptiProfiler plots the *extended* log-ratio profile defined in (5.2) (see Remark 5.2 in Section 5), and in Section 5 we show that (extended) log-ratio profiles are equivalent to performance profiles when comparing two solvers.

## 4.2 History-based and output-based costs

In this subsection, we define the absolute cost $t_{p,s}$ for $s \in \mathcal{S}$ to solve $p \in \mathcal{P}$. This requires us to first establish how we measure constraint violations and solution quality. Consider a general optimization problem $p \in \mathcal{P}$ in the form of (3.1), and define its $\ell_\infty$-constraint violation function $v_\infty : \mathbb{R}^n \to \mathbb{R}$ by

$$\begin{aligned} v_\infty(x) \;=\; \max\big\{ & \|(x_\mathrm{l} - x)^+\|_\infty, \|(x - x_\mathrm{u})^+\|_\infty, \\ & \|(A_\mathrm{ub} x - b_\mathrm{ub})^+\|_\infty, \|A_\mathrm{eq} x - b_\mathrm{eq}\|_\infty, \\ & \|(c_\mathrm{ub}(x))^+\|_\infty, \|c_\mathrm{eq}(x)\|_\infty \big\}. \end{aligned}$$

Using this constraint violation function, we construct a merit function $\varphi : \mathbb{R}^n \to \mathbb{R}$ that balances objective value and constraint satisfaction,

$$\varphi(x) \;=\; \begin{cases} f(x) & \text{if } v_\infty(x) \le v_1, \\ f(x) + \mu(v_\infty(x) - v_1) & \text{if } v_1 < v_\infty(x) \le v_2, \\ \infty & \text{otherwise,} \end{cases}$$

where $\mu$, $v_1$, and $v_2$ are nonnegative constants. In OptiProfiler, we set $\mu = 10^5$ by default, and

$$\begin{aligned} v_1 &\;=\; \max\{10^{-5}, v_0\}, \\ v_2 &\;=\; \min\{10^{-2}, 10^{-10} \max\{1, v_0\}\}, \end{aligned}$$

where $v_0$ is the $\ell_\infty$-constraint violation at the initial guess $x_0$. For unconstrained problems, the merit function $\varphi$ will reduce to the objective function $f$.

To determine whether a solver has successfully solved a problem, we employ the convergence test proposed in [15]. A point $x$ passes the convergence test on problem $p$ with a tolerance $\tau \in [0, 1]$ if

$$\varphi(x) \;\le\; \varphi^* + \tau(\varphi(x_0) - \varphi^*), \tag{4.3}$$

where $x_0$ is the initial guess and $\varphi^*$ is a reference lower bound of the merit function $\varphi$. We set $\varphi^*$ to the least value of $\varphi$ achieved by any solver in $\mathcal{S}$ on problem $p$ in this experiment. We provide an option called `run_plain` to determine whether to also run the experiment with the `plain` feature (see Section 3); if enabled, we set $\varphi^*$ to the minimum of the reference values obtained under the chosen feature and under `plain`.

Note that when testing derivative-based solvers, it can be appealing to set $\varphi^*$ to the value at a known minimizer, the least value achieved so far by any solver, or the least value achieved by a reliable reference solver. However, according to [15], these choices may not be appropriate for benchmarking derivative-free solvers, since it is possible that no solver in $\mathcal{S}$ passes the convergence test and all profiles lose their effectiveness.

Given these definitions, we now introduce two methods for measuring the absolute cost $t_{p,s}$.

- **History-based cost:** We set $t_{p,s}$ to the number of function evaluations that solver $s$ requires to first reach a point passing the convergence test (4.3) on problem $p$ (set to $\infty$ if solver $s$ never reaches such a point). This approach is the one commonly used in the literature [15].
- **Output-based cost:** We set $t_{p,s}$ as

$$t_{p,s} = \begin{cases} \mathrm{eval}_{p,s} & \text{if } x_{p,s}^{\mathrm{out}} \text{ passes the convergence test (4.3)}, \\ \infty & \text{otherwise}, \end{cases} \tag{4.4}$$

  where $\mathrm{eval}_{p,s}$ is the total number of function evaluations of solver $s$ used while solving problem $p$, and $x_{p,s}^{\mathrm{out}}$ is the output solution on problem $p$ by solver $s$.

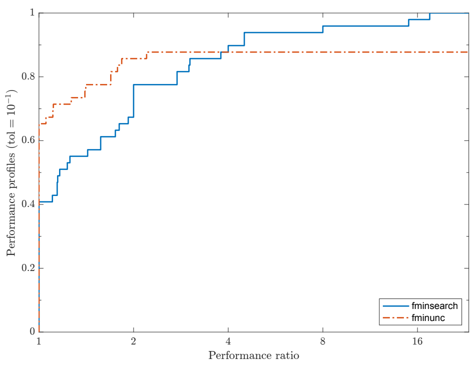
The history-based cost reflects how quickly a solver finds a solution that meets the desired accuracy, regardless of its stopping rule, and thus highlights intrinsic search efficiency. In contrast, the output-based cost accounts for the solver's entire run, including its stopping criterion. This metric evaluates not only the solver's ability to find good solutions, but also its effectiveness in deciding when to terminate. Together, these two costs provide a comprehensive view of both the solver's search capability and its practical usability, which is particularly useful for the design and analysis of optimization algorithms.

Note that in OptiProfiler, we enforce a simple budget-based stopping mechanism. We set a per-problem evaluation budget `maxfun` based on the problem dimension through the option `max_eval_factor`. If a solver exceeds `maxfun`, we set the objective function value and/or the nonlinear constraints to their values at the final evaluated point (so the solver may still stop by itself). If the solver does not stop after 2 `maxfun` evaluations, we terminate the run. For history-based profiles, we only use the evaluation history truncated at `maxfun`. For output-based profiles, we use the solver's returned output; if the solver fails to return, we treat the initial point as its output.

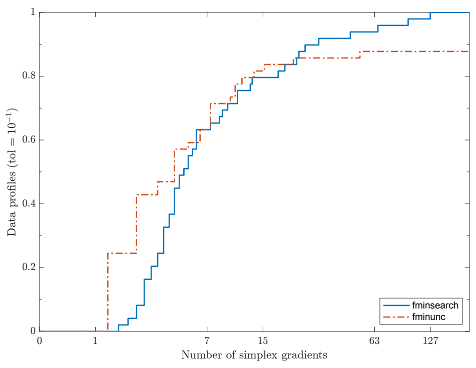
## 4.3 Profiles generated by OptiProfiler

In this subsection, we describe how OptiProfiler generates and plots performance, data, and log-ratio profiles in different settings.

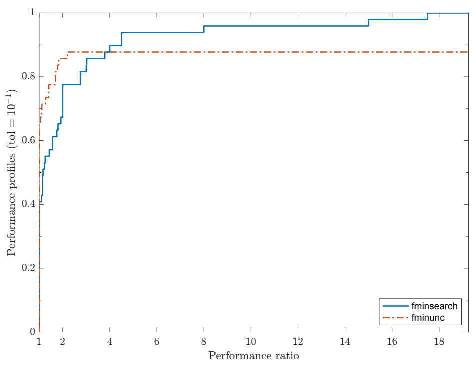
We provide an option called `semilogx` to determine whether the horizontal axis of performance and data profiles uses a logarithmic scale (base 2); the default value is `true`. Since performance profiles and data profiles are defined on $\mathbb{R}_+$, we truncate the curves on the right. Specifically, we take the $x$-coordinate of the last jump across all curves and enlarge it slightly (multiplying by 1.1) to obtain the right truncation point. If the horizontal axis uses a logarithmic scale, the truncation point is determined using the log-transformed $x$-coordinates. Figure 3a and Figure 3b show the history-based performance and data



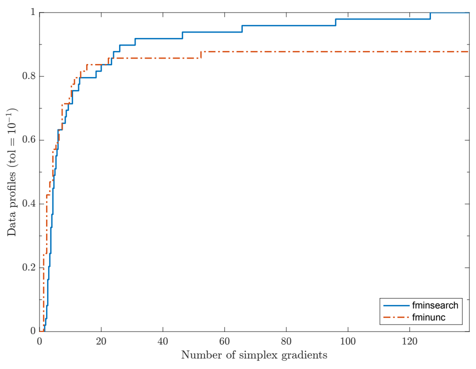
(a) History-based performance profile with $\tau = 0.1$ and with semilogx



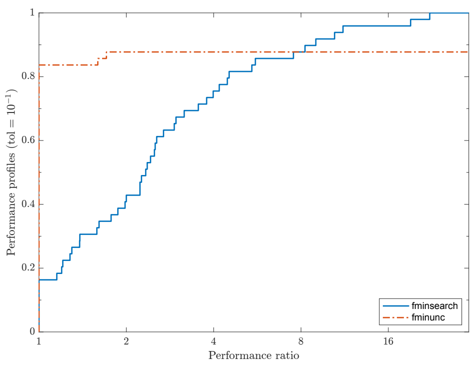
(b) History-based data profile with $\tau = 0.1$ and with semilogx



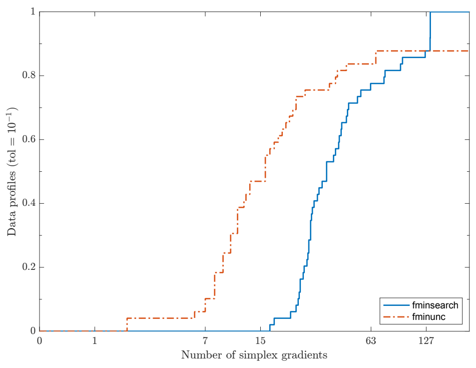
(c) History-based performance profile with $\tau = 0.1$ and without semilogx



(d) History-based data profile with $\tau = 0.1$ and without semilogx



(e) Output-based performance profile with $\tau = 0.1$ and with semilogx



(f) Output-based data profile with $\tau = 0.1$ and with semilogx

Figure 3: History-based and output-based performance and data profiles with and without semilogx

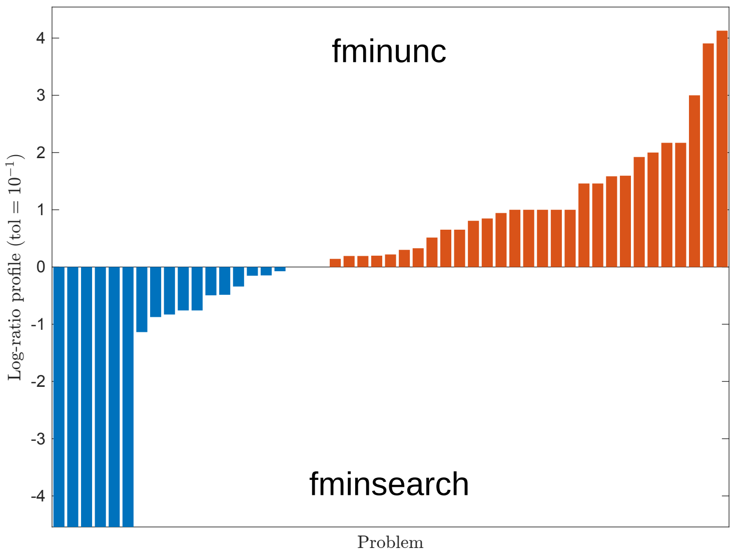


Figure 4: History-based log-ratio profile with $\tau = 0.1$

profiles with tolerance $\tau = 0.1$ and log-scale on the horizontal axis, while Figure 3c and Figure 3d show the corresponding plots without log-scale.

For log-ratio profiles, OptiProfiler plots the *extended* log-ratio profile defined in Remark 5.2, which explicitly includes the problems in $\mathcal{E}$ (i.e., problems that both solvers fail to solve); see also Figure 7 for how these additional cases appear as light bars.

Some (extended) log-ratio values may be $\infty$ or $-\infty$. In this case, we identify the maximum absolute value among all finite log-ratios, multiply it by 1.1, and use it to truncate the infinite values while preserving their signs. Figure 4 shows the history-based log-ratio profile of the previous example with $\tau = 0.1$. Note that history-based and output-based profiles need not look similar. Intuitively, history-based profiles emphasize solver efficiency, whereas output-based profiles also reflect the effect of stopping criteria. We compare Figure 3e and Figure 3f with Figure 3a and Figure 3b to illustrate this point.

Moreover, there is an option called `n_runs`. It is used to specify the number of runs of the experiment, which is always essential when the feature is stochastic such as `noisy`. When `n_runs` is greater than 1, we will draw the average curve of the performance profiles or data profiles across all runs. Besides, we will also use lighter colors to draw the error bars. The default error bar uses the point-wise minimum and maximum values of the performance profiles or data profiles across all runs. We still use the previous example. Figure 5 and Figure 6 show the history-based performance profile and the data profile with convergence tolerance $\tau = 0.1$ under the `noisy` feature with `n_runs` $= 5$. For log-ratio profiles, we do not average curves across runs. Instead, we treat the same problem in different runs as distinct problems and draw the log-ratio profile for the enlarged problem

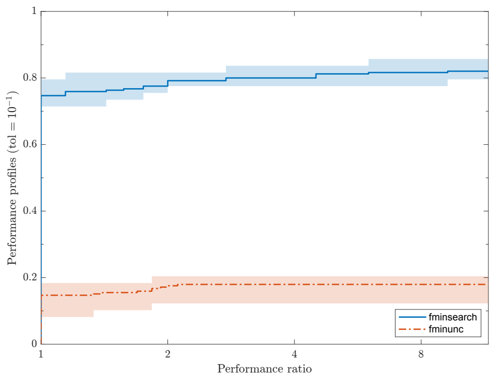


Figure 5: History-based performance profile with $\tau = 0.1$ and $\texttt{n\_runs} = 5$

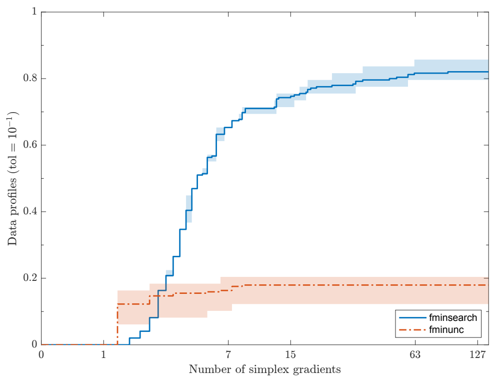


Figure 6: History-based data profile with $\tau = 0.1$ and $\texttt{n\_runs} = 5$

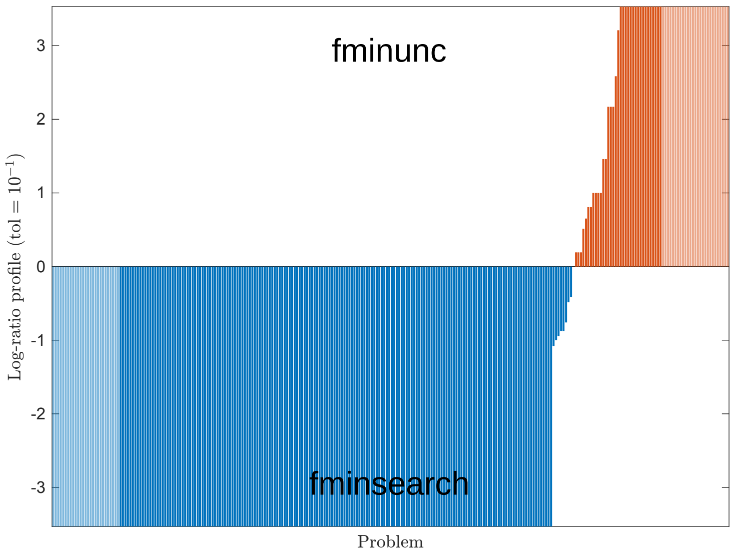


Figure 7: History-based log-ratio profile with $\tau = 0.1$ and $\texttt{n\_runs} = 5$

library. Figure 7 shows the history-based (extended) log-ratio profile of the previous example.

In Figure 7, the lighter bars correspond to the additional problems introduced by the *extended* definition (Remark 5.2). In particular, they represent runs in which both solvers fail to solve the problem (i.e., the run belongs to $\mathcal{E}$); these runs are displayed as light bars at the two extremes of the plot, effectively adding the same "both-failed" cases to both solvers.

## 4.4 Scoring solvers based on profiles

To quantitatively compare solvers, OptiProfiler computes the area under the curve (AUC) of a given profile. Mathematically, let $\phi : [a, b] \to \mathbb{R}_+$ denote a profile function (e.g., a performance profile or data profile) defined on a truncated domain $[a, b]$. The AUC is defined as

$$\mathrm{AUC}(\phi) = \int_a^b \phi(x)\,\mathrm{d}x. \tag{4.5}$$

For performance profiles and data profiles, the AUC directly serves as the solver's reference score, reflecting overall performance (with the convention that a larger AUC indicates better performance). For log-ratio profiles, we interpret the lower envelope of the shaded region below the bar plot as the negative of the profile function for the first solver, and the upper envelope of the shaded region above the bar plot as the profile function for the second solver. We then compute the corresponding AUC values as their scores (equivalently, the shaded areas in the bar plots).

A few implementation details are as follows.

- For performance and data profiles, the upper limit of the AUC integral is set to the same right truncation point used for plotting, i.e., 1.1 times the $x$-coordinate of the last jump across all curves.
- Numerical integration of the AUC is accurate since the profiles are piecewise constant.
- If the horizontal axis uses a logarithmic scale, the AUC is computed with respect to the log-transformed $x$-coordinates; an optional weight function can be used in the integration.
- For multiple runs, the score of a solver in a performance/data profile is calculated using the AUC of the average profile across all runs. In contrast, the scores of solvers in log-ratio profiles remain the same, as we treat the same problem in different runs as distinct problems in the log-ratio profiles.
- The default score of a solver in one experiment is the average score of all the history-based performance profiles across all tolerances; normalized scores (dividing by the maximum across solvers) are also available.

## 5 Equivalence between performance profile and log-ratio profile

In this section, we show that performance profiles and log-ratio profiles are equivalent when there are only two solvers. Recall the definition of the log-ratio profile:

$$\{\lambda_1, \ldots, \lambda_{|\mathcal{P}|-|\mathcal{E}|}\} \;=\; \mathrm{sort}\left(\{l_p : p \in \mathcal{P}\backslash\mathcal{E}\}\right).$$

We then define real-valued extensions of the log-ratio profile as

$$\begin{aligned} \tilde{\lambda}_{s_1}(t) &= \max\{0, \lambda_{\lceil(|\mathcal{P}|-|\mathcal{E}|)t\rceil}\}, \\ \tilde{\lambda}_{s_2}(t) &= \max\{0, -\lambda_{\lfloor(|\mathcal{P}|-|\mathcal{E}|)(1-t)\rfloor+1}\}, \end{aligned} \tag{5.1}$$

for $t \in (0, 1]$, where $\lceil\cdot\rceil$ is the ceiling function and $\lfloor\cdot\rfloor$ is the floor function. We also define $\tilde{\rho}_s(\alpha) = \rho_s(2^\alpha)$ for $s \in \{s_1, s_2\}$ and $\alpha \geq 0$.

**Theorem 5.1.** *The log-ratio profile can be viewed as an inverse function of the performance profile; that is, for $s \in \{s_1, s_2\}$ and $\alpha > 0$,*

$$\tilde{\rho}_s(\alpha) \;=\; \frac{|\mathcal{P}| - |\mathcal{E}|}{|\mathcal{P}|}\tilde{\lambda}_s^{-1}(\alpha),$$

*where $\tilde{\lambda}_s^{-1}(\alpha) = \sup\{t \in (0, 1] : \tilde{\lambda}_s(t) < \alpha\}$.*

**Proof.** We prove only the case $s = s_1$; the case $s = s_2$ follows similarly. For any $\alpha > 0$, we have

$$\begin{aligned} \frac{|\mathcal{P}|}{|\mathcal{P}| - |\mathcal{E}|}\tilde{\rho}_{s_1}(\alpha) &= \frac{1}{|\mathcal{P}| - |\mathcal{E}|}\left|\left\{p \in \mathcal{P} : \frac{t_{p,s_1}}{\min\{t_{p,s_1}, t_{p,s_2}\}} \leq 2^\alpha\right\}\right| \\ &= \frac{1}{|\mathcal{P}| - |\mathcal{E}|}\left(\left|\left\{p \in \mathcal{P} : \frac{t_{p,s_1}}{t_{p,s_2}} \leq 2^\alpha \text{ and } t_{p,s_1} > t_{p,s_2}\right\}\right|\right. \\ &\quad + \left|\left\{p \in \mathcal{P} : \frac{t_{p,s_1}}{t_{p,s_1}} \leq 2^\alpha \text{ and } t_{p,s_1} < t_{p,s_2}\right\}\right| \\ &\quad \left. + \left|\{p \in \mathcal{P} : t_{p,s_1} = t_{p,s_2} < \infty\}\right|\right). \end{aligned}$$

By the definition of log-ratio profile and $\mathcal{E}$, we have

$$\begin{aligned} \frac{|\mathcal{P}|}{|\mathcal{P}| - |\mathcal{E}|}\tilde{\rho}_{s_1}(\alpha) &= \frac{1}{|\mathcal{P}| - |\mathcal{E}|}\Big( |\{k = 1, \ldots, |\mathcal{P}| - |\mathcal{E}| : 0 < \lambda_k \leq \alpha\}| \\ &\quad + |\{k = 1, \ldots, |\mathcal{P}| - |\mathcal{E}| : \lambda_k < 0\}| \\ &\quad + |\{k = 1, \ldots, |\mathcal{P}| - |\mathcal{E}| : \lambda_k = 0\}|\Big) \\ &= \frac{1}{|\mathcal{P}| - |\mathcal{E}|}\left|\{k = 1, \ldots, |\mathcal{P}| - |\mathcal{E}| : \lambda_k \leq \alpha\}\right|. \end{aligned}$$

Since the log-ratio values are sorted in ascending order, we have

$$\begin{aligned}
\frac{|\mathcal{P}|}{|\mathcal{P}|-|\mathcal{E}|}\tilde{\rho}_{s_1}(\alpha) &= \frac{1}{|\mathcal{P}|-|\mathcal{E}|}\sup\{k=1,\dots,|\mathcal{P}|-|\mathcal{E}| : \lambda_k \le \alpha\} \\
&= \sup\left\{t\in(0,1] : \lambda_{\lceil(|\mathcal{P}|-|\mathcal{E}|)t\rceil} < \alpha\right\} \\
&= \sup\left\{t\in(0,1] : \max\{0,\lambda_{\lceil(|\mathcal{P}|-|\mathcal{E}|)t\rceil}\} < \alpha\right\} \\
&= \sup\left\{t\in(0,1] : \tilde{\lambda}_{s_1}(t) < \alpha\right\} \\
&= \tilde{\lambda}_{s_1}^{-1}(\alpha).
\end{aligned}$$

□

**Remark 5.1.** We can also prove the reverse direction of Theorem 5.1, that is,

$$\tilde{\lambda}_s(t) = \tilde{\rho}_s^{-1}\left(\frac{|\mathcal{P}|-|\mathcal{E}|}{|\mathcal{P}|}t\right),$$

if we define $\tilde{\rho}_s^{-1}(t)=\sup\{\alpha\ge 0 : \tilde{\rho}_s(\alpha)\le t\}$ for $t\in(0,1]$. Note that $\tilde{\rho}$ is a nondecreasing right-continuous step function and $\tilde{\lambda}$ is a nondecreasing left-continuous step function, so the inverse function here is different from that in Theorem 5.1.

**Remark 5.2.** For additional clarity, we define the extended log-ratio profile by

$$\{\hat{\lambda}_1,\dots,\hat{\lambda}_{|\mathcal{P}|+|\mathcal{E}|}\} = \operatorname{sort}\left(\{\underbrace{-\infty,\dots,-\infty}_{|\mathcal{E}|}\}\cup\{l_p : p\in\mathcal{P}\}\right), \tag{5.2}$$

with the convention that $\log_2(\infty/\infty)=\infty$. After modifying the definition of $\tilde{\lambda}_s$ to

$$\begin{aligned}
\tilde{\lambda}_{s_1}(t) &= \max\{0,\lambda_{|\mathcal{E}|+\lceil|\mathcal{P}|t\rceil}\}, \\
\tilde{\lambda}_{s_2}(t) &= \max\{0,-\lambda_{\lfloor|\mathcal{P}|(1-t)\rfloor+1}\},
\end{aligned} \tag{5.3}$$

we can establish a more direct equivalence between performance profile and extended log-ratio profile as

$$\tilde{\rho}_s(\alpha) = \tilde{\lambda}_s^{-1}(\alpha). \tag{5.4}$$

Figure 8 illustrates the equivalence between performance profiles and extended log-ratio profiles. In the left panel, the blue (resp., red) boxed region and the dark blue (resp., red) curve correspond to the performance profile of solver $s_1$ (resp., $s_2$) after reflection with respect to the vertical axis. If we replace the log-ratio by the ratio (equivalently, use a log-scaled horizontal axis for the performance profile), then the area above the curve of the performance profile of solver $s_1$ (resp., $s_2$) corresponds to the shaded area of the extended log-ratio profile of solver $s_2$ (resp., $s_1$). The right panel simply shows the performance

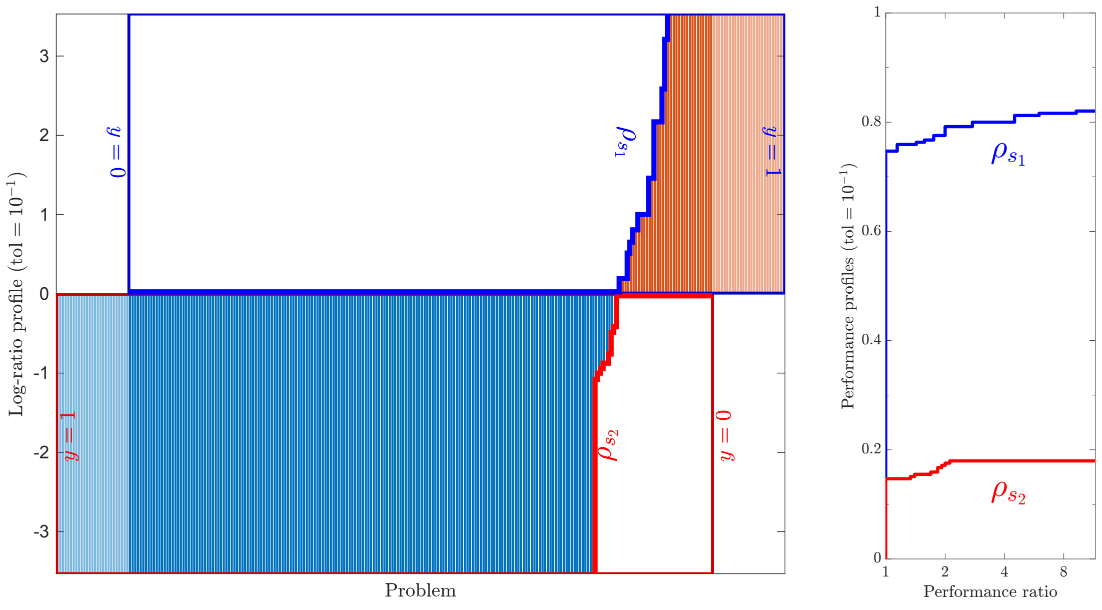


Figure 8: Graphical illustration of the equivalence between performance profile and extended log-ratio profile

profiles from the left panel. Note that both log-ratio profiles and extended log-ratio profiles are equivalent to performance profiles when $|\mathcal{E}|$ is known. We plot extended log-ratio profiles in OptiProfiler because their shaded areas relate directly to the AUC of performance profiles.

**Remark 5.3.** If we run the experiment multiple times (`n_runs` $> 1$), then the average performance profile is equivalent to the extended log-ratio profile. The proof is similar to the one above and is omitted here.

## 6 Experimental results

In this section, we present and analyze experimental results obtained using OptiProfiler. We first demonstrate how to benchmark DFO solvers with a simple example. We then show how OptiProfiler can be used to tune the hyperparameters of DFO solvers.

## 6.1 Demonstration of benchmarking DFO solvers

To demonstrate the core capabilities of OptiProfiler, we select three representative unconstrained DFO solvers and benchmark them on several testing environments. The solvers are listed below.

- NEWUOA (MATLAB implementation in PRIMA [20]),
- the Nelder–Mead simplex method (`fminsearch` in MATLAB),
- a finite-difference BFGS method (`fminunc` in MATLAB when the gradient is not provided).

The test problems are all unconstrained problems from both the S2MPJ and MatCUTEst problem libraries with dimensions ranging from 1 to 50. To assess solver performance across various scenarios, we consider six different features with their default settings: `plain`, `noisy`, `perturbed_x0`, `truncated`, `linearly_transformed`, and `random_nan`.

For each feature, we report both history-based and output-based performance profiles and data profiles under two representative convergence tolerances: a low-accuracy setting ($\tau = 0.1$) and a high-accuracy setting ($\tau = 10^{-10}$). For each feature, the results are presented in one row with eight plots, ordered from left to right as follows. Four performance profiles (history-based, $\tau = 0.1$; history-based, $\tau = 10^{-10}$; output-based, $\tau = 0.1$; output-based, $\tau = 10^{-10}$), followed by four data profiles in the same order.

The experiments are conducted on GitHub Actions using Ubuntu 24.04 and MATLAB R2024b. Figure 9 shows the profiles for the six features. Note that the x-axis and y-axis labels are omitted for clarity. We find that the performance of NEWUOA (the blue curve) is generally better than that of the other two solvers, which is consistent with the general understanding (e.g., [15]) that NEWUOA is a very competitive DFO solver.

## 6.2 Tuning hyperparameters of DFO solvers

In this subsection, we use a simple example to demonstrate that OptiProfiler can be used to tune the hyperparameters of DFO solvers in a few lines of code.

Consider Algorithm 6.1, which is the simplified probabilistic direct search (PDS) algorithm in [9]. Algorithm 6.1 has two hyperparameters, the shrinking factor $\theta \in (0, 1)$ and the expanding factor $\gamma \geq 1$, with default values $\theta = 0.5$ and $\gamma = 2$. Note that in our implementation, PDS terminates if either the number of iterations reaches 1000 or the step size becomes smaller than $10^{-8}$. Our goal is to optimize these hyperparameters to improve the solver's performance on all unconstrained problems with dimension at most 5 in the default library, using AUC of the history-based performance profile with tolerance $\tau = 10^{-4}$ as the metric. We tune the hyperparameters by maximizing the difference in

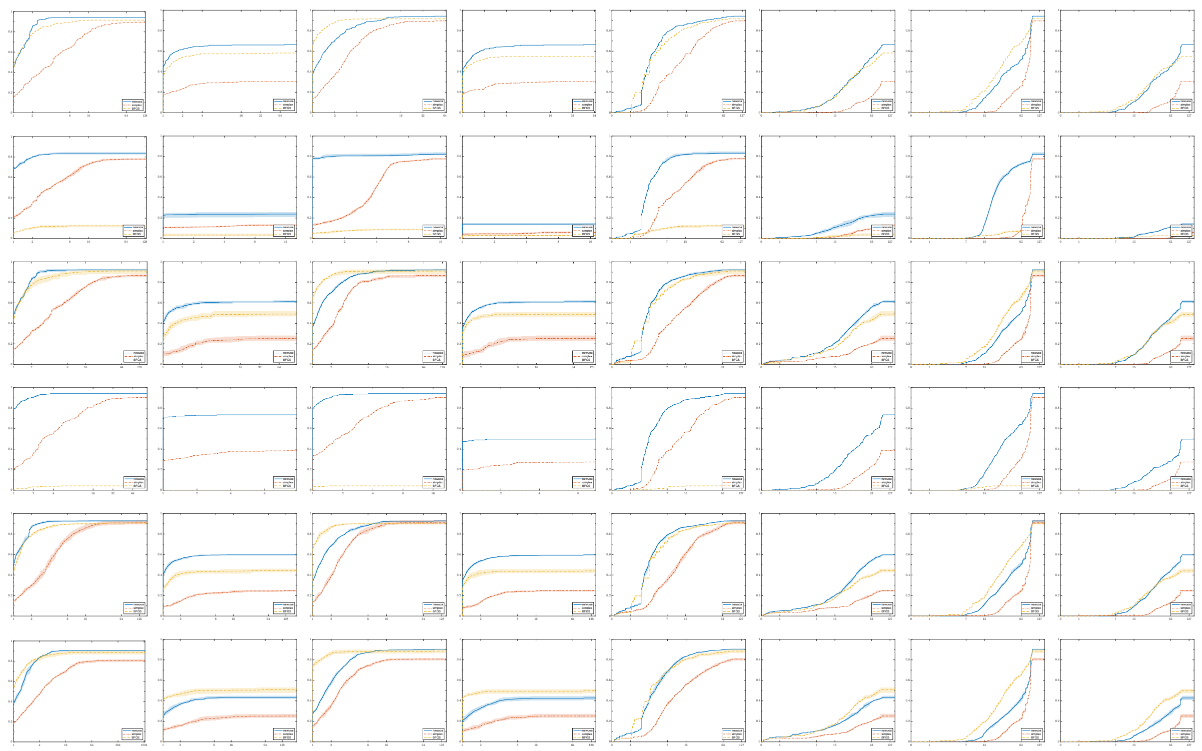


Figure 9: Profiles with different features. From top to bottom, the features are `plain`, `noisy`, `perturbed_x0`, `truncated`, `linearly_transformed`, and `random_nan`.

**Algorithm 6.1** Simplified probabilistic direct search

```
Select $x_0 \in \mathbb{R}^n$, $\alpha_0 = 1$, $\theta \in (0, 1)$, $\gamma \in [1, \infty)$.
For $k = 0, 1, 2, \ldots$, do the following.
  1. Generate $\mathfrak{d}_k \in \mathbb{R}^n$ following the uniform distribution on the unit sphere in $\mathbb{R}^n$.
  2. If there exists $d_k \in \{\mathfrak{d}_k, -\mathfrak{d}_k\}$ such that
         $$f(x_k) - f(x_k + \alpha_k d_k) \; > \; 10^{-3}\alpha_k^2,$$
     then set $x_{k+1} = x_k + \alpha_k d_k$, $\alpha_{k+1} = \gamma\alpha_k$; otherwise, set $x_{k+1} = x_k$, $\alpha_{k+1} = \theta\alpha_k$.
```

AUC between PDS with tuned hyperparameters and the one with default hyperparameters. Mathematically speaking, we are going to solve the following optimization problem:

$$\begin{aligned} \min_{\theta,\gamma\in\mathbb{R}} \quad & \mathcal{L}(\theta,\gamma) \overset{\text{def}}{=} \mathrm{AUC}(\rho_{s_{\text{tuned}}}) - \mathrm{AUC}(\rho_{s_{\text{default}}}) \\ \text{s.t.} \quad & \theta \in (0,1), \\ & \gamma \in [1,\infty), \end{aligned} \tag{6.1}$$

where $\rho_{s_{\text{tuned}}}$ and $\rho_{s_{\text{default}}}$ are the truncated performance profiles of PDS with tuned hyperparameters and with default ones, respectively. Instead of solving Problem (6.1) directly, we will simply solve the bound-constrained optimization problem (6.2).

$$\begin{aligned} \min_{x,y\in\mathbb{R}} \quad & \mathcal{L}(2^{-x}, 2^{y}) \\ \text{s.t.} \quad & x \geq \epsilon, \\ & y \geq 0, \end{aligned} \tag{6.2}$$

where $\epsilon$ is a small positive number. Regarding $\mathcal{L}$ as a black-box function, we use BOBYQA [17, 18, 20] to solve Problem (6.2). The experiment is conducted on a system running Ubuntu 22.04.5 LTS with MATLAB R2023b, equipped with a 12th Gen Intel Core i7-12700 processor (20 cores) and 62 GB of RAM. We set $\epsilon$ to eps in double precision and set the maximum number of function evaluations of BOBYQA to 100. For simplicity, we also fix the seed of the random number generator inside PDS so that the loss function $\mathcal{L}$ is deterministic. The following is the complete code we use, except for the implementation of Algorithm 6.1, which is named `pds` in the script.

```matlab
% Tune hyperparameters of Algorithm PDS.

x0 = [-1; 1]; % Initial guess.
lb = [-Inf; 0]; % Lower bound.
ub = [-eps; Inf]; % Upper bound.
options.maxfun = 100; % BOBYQA Maximum function evaluations.
x = bobyqa(@tuning_loss, x0, lb, ub, options) % Call BOBYQA.

function f = tuning_loss(x)
    theta = 2^x(1);
    gamma = 2^x(2);
    solvers = cell(1, 2);
    solvers{1} = @(fun, x0) pds(fun, x0, 0.5, 2);
    solvers{2} = @(fun, x0) pds(fun, x0, theta, gamma);
    options.maxdim = 5;
    options.scoring_fun = @(x) x(:, 4, 1, 1);
```

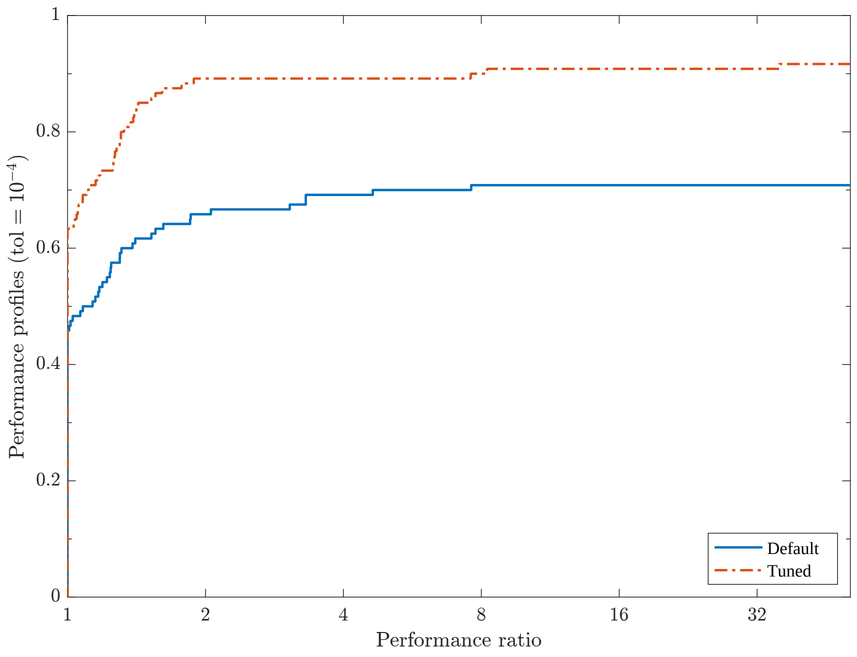


Figure 10: History-based performance profile with $\tau = 10^{-4}$ of Algorithm 6.1 with default and tuned hyperparameters.

```
    scores = benchmark(solvers, options);
    f = scores(1) - scores(2);
end
```

After 37 function evaluations, BOBYQA returns the solution $(-1.4251, 1.7360)^{\mathsf{T}}$, which corresponds to $\theta \approx 0.3724$ and $\gamma \approx 3.3312$. Figure 10 shows the final history-based performance profile with tolerance $10^{-4}$ for Algorithm 6.1 with default and tuned hyperparameters, demonstrating a significant performance improvement after tuning.

## 7 Conclusion and future extensions

In this paper, we introduced OptiProfiler, an extensible and user-friendly benchmarking platform for optimization solvers, with a current focus on derivative-free optimization. OptiProfiler automates the generation of performance profiles, data profiles, and log-ratio profiles, and supports a variety of benchmarking tasks with multiple features and problem libraries. Our experiments demonstrate that OptiProfiler is a powerful tool for benchmarking DFO solvers by providing detailed profiles and reference scores.

There are several directions for future development of OptiProfiler.

- **Broader language support.** Currently, OptiProfiler is implemented in MATLAB. We plan to extend support to other major programming languages, including Python, C, and Julia, to make the platform accessible to a wider community.
- **Benchmarking gradient-based solvers.** While the current focus is on derivative-free optimization, we aim to support benchmarking of first-order optimization solvers as well. This will involve creating suitable benchmarks and measuring time-based cost for first-order methods.
- **Multi-objective optimization.** We intend to extend OptiProfiler to support benchmarking of multi-objective optimization algorithms following the state-of-the-art benchmarks and cost definitions.
- **Online benchmarking and cloud support.** In addition to local deployment, we plan to develop an online benchmarking platform that leverages cloud computing resources, enabling users to submit solvers and obtain benchmarking results conveniently and efficiently.

These extensions will further enhance the versatility and impact of OptiProfiler, making it a comprehensive benchmarking platform for the optimization community.

## References

[1] C. Audet and W. Hare. *Derivative-Free and Blackbox Optimization.* Springer, Cham, 2017.

[2] C. Audet, W. Hare, and C. Tribes. Benchmarking constrained, multi-objective and surrogate-assisted derivative-free optimization methods. Technical Report G-2025-36, Groupe d'études et de recherche en analyse des décisions, 2025.

[3] V. Beiranvand, W. Hare, and Y. Lucet. Best practices for comparing optimization algorithms. *Optim. Eng.*, 18:815–848, 2017.

[4] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*, volume 8 of *MOS-SIAM Ser. Optim.* SIAM, Philadelphia, 2009.

[5] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91:201–213, 2002.

[6] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEr and SifDec: A constrained and unconstrained testing environment, revisited. *ACM Trans. Math. Software*, 29:373–394, 2003.

[7] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Comput. Optim. Appl.*, 60:545–557, 2015.

[8] N. I. M. Gould and J. Scott. A note on performance profiles for benchmarking software. *ACM Trans. Math. Software*, 43:1–5, 2016.

[9] S. Gratton, C. W. Royer, L. N. Vicente, and Z. Zhang. Direct search based on probabilistic descent. *SIAM J. Optim.*, 25:1515–1541, 2015.

[10] S. Gratton and Ph. L. Toint. S2MPJ and CUTEst optimization problems for Matlab, Python and Julia. *arXiv:2407.07812*, 2024.

[11] N. Hansen, A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff. COCO: a platform for comparing continuous optimizers in a black-box setting. *Optim. Methods Softw.*, 36:114–144, 2021.

[12] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright. Convergence properties of the Nelder–Mead simplex method in low dimensions. *SIAM J. Optim.*, 9:112–147, 1998.

[13] J. Larson, M. Menickelly, and S. M. Wild. Derivative-free optimization methods. *Acta Numer.*, 28:287–404, 2019.

[14] J. L. Morales. A numerical study of limited memory BFGS methods. *Appl. Math. Lett.*, 15:481–487, 2002.

[15] J. J. Moré and S. M. Wild. Benchmarking derivative-free optimization algorithms. *SIAM J. Optim.*, 20:172–191, 2009.

[16] J. A. Nelder and R. Mead. A simplex method for function minimization. *Comput. J.*, 7:308–313, 1965.

[17] M. J. D. Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical Report DAMTP 2009/NA06, Department of Applied Mathematics and Theoretical Physics, Cambridge University, Cambridge, 2009.

[18] T. M. Ragonneau and Z. Zhang. PDFO: A cross-platform package for Powell's derivative-free optimization solvers. *Math. Program. Comput.*, 16:535–559, 2024.

[19] H.-J. Shi, Q. Xuan, F. Oztoprak, and J. Nocedal. On the numerical performance of derivative-free optimization methods based on finite-difference approximations. *arXiv:2102.09762*, 2021.

[20] Z. Zhang. PRIMA: Reference Implementation for Powell's Methods with Modernization and Amelioration (version 0.7.3). https://www.libprima.net, 2023.

[21] Z. Zhang. MatCUTEst. `https://github.com/matcutest`, 2024.